Final Project: Knapsack Algorithms - 20 points

Important: This assignment is longer than the others so it is a good idea to get an early start. Late assignments or assignments not meeting the specifications of the assignment WILL NOT BE ACCEPTED.

You have just started work at the **Fly-By-Night Consulting Company** and they have been approached by a secretive potential client. Your boss recognizes the client's problem as the classic 0-1 knapsack problem. This may be a great opportunity for your company if you can find the right implementation to solve the client's problem. However, as usual, **there are issues**:

- Your company has no experience with Knapsack Algorithms and you are the only one with any time to quickly come up to speed.
- The client refuses to give you many details about the characteristics of the problem, they are worried about losing an advantage they feel they have in their market. Your task is to help your company recommend the right approach. To get ready, your boss has asked you to explore different algorithmic approaches and make sure the company understands their strengths and weaknesses.

You are to explore four approaches to solving the 0-1 Knapsack Problem **using Java** (full enumeration/brute force, greedy, dynamic programming, and branch-and bound). To make sure you understand the tradeoffs between the different algorithms you must implement the algorithms and run them on some test data to compare their running time and solution accuracy. (You may look at other code, **but you must document its source, and write the implementation yourself. You may not copy and paste code that is not yours!**)
Each of the four approaches should be implemented in a **separate method or class** that is easy to read and understand. **Generally** methods should be less than a page long. White space should be used sparingly to make it easy to read!

Step 0: Ensure you can read in the problem correctly
A knapsack problem instance will be in a text file written in the following format: C is the capacity and is an integer, N is the number of items, each item is preceded by its index that will serve as a unique identifier. The values and weights are also positive integers.

```
N
1   v1    w1
2   v2    w2
.   .     .
.   .     .
N   vN    wN
C
```

Full Enumeration

The 0-1 problem can be solved by a full enumeration of the search space. A binary string of length *N* represents a subset and thus valid candidate solution. A method should loop through every possible binary string of length *N* and evaluates the value and weight of each of these solutions.

Output Upon termination, the program **must output the solution the following format**:

```
Using Brute force the best feasible solution found: Value <value>, Weight <weight>
<item> <item> <item> ...
```

where the items are the indexes of the items to be placed in the knapsack, **items must be listed in ascending index order using the indices for the items specified in the file beginning with index 1**.

E.g.

```
Using Brute force the best feasible solution found:  Value 234, Weight 17
1 3 7 12...
```

Greedy Search

Greedy search can be applied to the 0/1 Knapsack problem, but it is not guaranteed to give an optimal solution. Remember that a greedy search has the form:

```
While solution not constructed
  Select best remaining element and try adding it into the knapsack
```

In your implementation, first sort the items by **the criterion used**. **You must have comments that indicate your criterion for choosing the next item to include.**

Output Upon termination, the program should output the following:

```
Greedy solution (not necessarily optimal):  Value <value>, Weight <weight>
<item> <item> <item> ...
```

items listed in ascending index order as specified for full enumeration.

Dynamic Programming

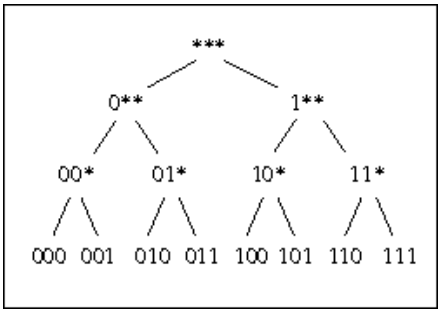
Output Upon termination, the program should output the following:

```
Dynamic Programming solution:  Value <value>, Weight <weight>
<item> <item> <item> ...
```

items listed in ascending index order as specified for full enumeration.

Branch-and-Bound

You will recall that a branch-and-bound approach is based on a tree search. Each node of the tree is a partial solution, with some solution components (items) determined, and some not. At the leaves of the tree are complete solutions. A partial solution can be represented as a string such as 011011*****, where the ***s** represent the parts of the solution that are not specified. (There is always a block of 0s and 1s followed by a block of *s). All possible ways of filling in the remaining bits of the solution will be below this node in the tree. Here is a search tree for a problem with three items:



The Branch-and-Bound approach is to visit some of the internal nodes, but to prune off the subtrees of nodes where an optimal solution *cannot be*, thus reducing the number of nodes to be visited. To dismiss a node without expanding it, we use a bound function. **One option is the *fractional knapsack upper bound* discussed in Levitin on pages 436-438. Your program must** clearly document the fragment of code that gives the computation that you use for upper bound of a candidate solution.

Your branch-and-bound algorithm may use a best-first search strategy based on these bound values. If you choose to do this, then every time you program generates a partial solution that is feasible, it will calculate its bound value and then place the item in a **priority queue**. The item at the head of the queue (the one that will be served next) will be the partial solution with the best bound seen so far. **Other search strategies and bounding functions are possible and should be considered.** You should understand the pros and cons of whatever approach you use. You may be asked to explain it during the demo of your code

Output Upon termination, the program should output the following:

```
Using Branch and Bound the best feasible solution found:  Value <value>, Weight <weight>
<item> <item> <item>
```

items listed in ascending index order as specified for full enumeration.

Final Report

Your narrative report should be less than four typewritten pages – I will not accept longer reports and it must be in 12 point readable font with line spacing of roughly 1.2 to 1.4. The report should document the basics of each implementation, e.g. the basic data structures and brief description of the algorithmic details.

There should be a summary of the pros and cons of each approach. Also, a discussion of potential ways to improve each of the specific algorithms you have implemented. E.g. different data structures, different orderings of the data etc. You goal is to communicate your understanding of the tradeoffs between the different approaches to the company's management.

In the past some students have not written a report but only supply table (see below) and some bullet points without any explanation. This is will not earn you anywhere near full credit. The writeup is important it counts for up to half the points for this assignment! In the past students have also lost significant points for incorrect conclusions or generalizations.

Run greedy, dynamic programming, and branch-and-bound on the datasets easy.20.txt, easy50.txt, hard50.txt, easy.200.txt, and hard.200.txt. Run full enumeration on easy.20.txt only. You will need additional test cases. The results on these test cases are **not** indicative of the how your algorithms will behave in all situations. On some of these data files you may have to halt your branch-and-bound early and take the best solution value to that point. Thus, you **must** put a timer in your program to complete after a certain number of minutes and then print the results of your best solution found up to that point along with the time it ran. Your branch and bound is not required to complete on hard200.txt. Completion for easy200 and/or hard50.txt data sets can be challenging.

Record your results in a table shown below, recording the best value found and the total weight if your program completes or the value and weight when the program is terminated by you – this should be clearly indicated. In addition, **you must include in the table the time the program took to complete or how long the program was allowed to run**. (Note: For Branch and Bound some students experience long run times, **e.g. overnight**). In notes following the table document what you observed and learned.

	easy20	**easy50**	**hard50**	**easy200**	**hard200**
Enum	value(weight)	**Do not run**	**Do not run**	**Do not run**	**Do not run**
Greedy	value(weight)	value(weight)	value(weight)	value(weight)	value(weight)
Dyn Prog	value(weight)	value(weight)	value(weight)	value(weight)	value(weight)
B'n'B	value(weight)	value(weight)	value(weight)	value(weight)	value(weight)

At your demo you must submit the hardcopy writeup and programs specified before your demo!!

Deliverables: These are all due when you make your demo. You will get up to 2 additional points if you **successfully demo and make your final submission of all the deliverables in lab on Weds**.

0. Demo in lab, showing outputs for all the different algorithms. Submit the following in lab **and** electronically to PolyLearn. # 1 should be submitted as a pdf file, #2, #3, #4 are to be .java text files.
1. A typed report
2. Hardcopy of your method(s) that generate the greedy candidate solution.
3. Hardcopy of your method(s) that generate the dynamic programming solution.
4. Hardcopy of your method(s) that generate the branch and bound solution – this may be up to two pages.

The total number of pages you should hand in is 6-7. Do not have a cover page. The pages must be stapled and in the order indicated above with your name and calpoly username on the report. You are responsible for meeting these specifications. Only submissions that meet these specifications will be graded.